A Grammar-based Data-flow Analysis to Stop Deforestation

Morten Heine Sørensen

DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark. E-mail: rambo@diku.dk

Abstract. Wadler's deforestation algorithm removes intermediate data structures from functional programs, but is only guaranteed to terminate for treeless programs. Chin has shown how one can apply deforestation to all first-order programs: annotate non-treeless subterms and apply the extended deforestation algorithm which essentially leaves annotated subterms untransformed. We develop a new technique of putting annotations on programs. The basic idea is to compute a finite grammar which approximates the set of terms that the deforestation algorithm encounters. The technique extends Wadler's and Chin's in a certain sense.

1 Introduction

Modern functional programming languages like Miranda\(^1\) [Tur90] lend themselves to a certain elegant style of programming which exploits higher-order functions, lazy evaluation and intermediate data structures; Hughes [Hug90] gives illuminating examples. While this programming style makes it easy to read and write programs, it also results in inefficient programs.

Early work on automatic elimination of intermediate data structures from functional programs includes Turchin’s supercompiler [Tur80,Tur82] and Wadler’s listless transformer [Wad84,Wad85]. The latter eliminates intermediate lists, but later Wadler invented deforestation [Wad88,Fer88], which eliminates intermediate data structures in general, and proved that the deforestation algorithm terminates when applied to treeless programs.

A method for applying deforestation to all first-order programs was later described by Chin [Chi90,Chi94]. Inspired by Wadler’s blazed deforestation algorithm [Wad88] he invented an extended deforestation algorithm which essentially leaves annotated subterms untransformed. The problem remains to calculate annotations ensuring that application of the extended deforestation algorithm to the annotated program terminates; as few subterms as possible should be annotated. Chin essentially annotates non-treeless subterms.

This paper describes a new static analysis whose result can be used to ensure termination of deforestation in such a way that fewer annotations will be put on the program, compared to Chin’s technique.

The remainder is organized as follows. Section 2: the language we study. Section 3: the deforestation algorithm. Section 4: the two kinds of non-termination

\(^1\) Miranda is a trademark of Research Software Ltd.
that can arise during deforestation. Section 5: some notation for the analysis. Section 6: the idea of the analysis and how it works on the two examples from section 4. Section 7: the analysis technically. Section 8: an improvement of the basic method. Section 9: an explanation of the fact that our method extends the methods by Wadler and Chin; the improvement of Section 8 is necessary for this. Section 10: conclusion.

2 Language and notation

We study the same language as the one studied in [Fer88].

Definition 1 Object language. Let \( c, v, f, g \) range over constructor names, variable names, \( f \)-function names, and \( g \)-function names, respectively. Let \( t, p, d \) range over terms, patterns, and definitions, respectively.

\[
\begin{align*}
t &::= v | c t_1 \ldots t_n | f t_1 \ldots t_n | g t_0 \ldots t_n | \text{ let } v = t \text{ in } t' \\
p &::= c \: v_1 \ldots v_n \\
d &::= f \: v_1 \ldots v_n \leftarrow t \\
 &\quad | \quad g \: p_1 \: v_1 \ldots v_n \leftarrow t_1 \\
 &\quad | \quad \vdots \\
 &\quad | \quad g \: p_k \: v_1 \ldots v_n \leftarrow t_k
\end{align*}
\]

\( g \)-functions have one pattern matching argument, \( f \)-functions have none. We use \( h \) to range over functions which are either \( f \)- or \( g \)-functions.

We require that no variable occur more than once in a pattern and that all variables of a right hand side of a definition be present in the corresponding left hand side. To ensure uniqueness of reduction, we require from a program that each function have at most one definition and, in the case of a \( g \)-definition, that no two patterns \( p_i \) and \( p_j \) contain the same constructor.

The semantics for reduction of a variable-free term is lazy evaluation, like in Miranda.

As usual we state the deforestation algorithm by rule for rewriting terms. For this, we need some notation that allows us to pick a function call in a term and replace the call by the body of the function with arguments substituted for formal parameters. Since the deforestation algorithm basically simulates call-by-name evaluation, there is always a unique function call whose unfolding is forced. For instance, to find out which of \( g \)'s clauses to apply to \( g \:( f_1 \: t_1 ) \:( f_2 \: t_2 ) \) we are forced to unfold the call \( f_1 \: t_1 \). In the terminology below, the forced call \( f_1 \: t_1 \) is the redex and the surrounding term \( g \ [] \:( f_2 \: t_2 ) \) is the context. If the term is a variable or has an outermost constructor, it is an observable.